Marcus Alanen | Ivan Porres

Subset and Union Properties in Modeling Languages

Turku Centre *for* Computer Science

TUCS Technical Report
No 731, December 2005

Subset and Union Properties in Modeling Languages

Marcus Alanen
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14 A, 20520 Turku, Finland
marcus.alanen@abo.fi
Ivan Porres
Åbo Akademi University, Department of Computer Science
Lemminkäisenkatu 14 A, 20520 Turku, Finland
ivan.porres@abo.fi

TUCS Technical Report
No 731, December 2005

Abstract

This paper discusses the new property characteristics in the Meta Object Facility 2.0, namely subset and union properties. They are heavily used in the Unified Modeling Language 2.0 standard, but lack a formal definition. We give our understanding of the new characteristics by formalizing subsets and unions using substitutability as our criterion. We present basic operations to create and edit models that use subset and unions properties. These operations form the basis of a model repository component in a modeling tool and are required to support Unified Modeling Language 2.0 models.

Keywords: subsets, derived unions, metamodeling, MOF, UML

TUCS Laboratory
Software Construction Laboratory

1 Introduction

Software modeling languages are used to describe the syntax and semantics of software models, i.e., abstract descriptions of software systems. Modeling is a fundamental approach to problem solving in all engineering disciplines, including software engineering. Software modeling and software modeling languages have recently become accepted by the software industry thanks to initiatives such as the UML and the Model Driven Architecture [20] by the Object Management Group (OMG).

The OMG modeling standards are based on the concept of metamodeling. A metamodeling language is a computer language used to describe other modeling languages. Metamodeling aims to describe software and system models and modeling languages in an uniform way. Another feature of the OMG modeling standards is that the abstract syntax and concrete syntax of a language are defined independently. This separation is so strong that often the term model is used to describe the abstract syntax of an artifact, while the term diagram is used to describe its concrete syntax using a visual language.

The term metamodel denotes an artifact that defines the abstract syntax of a modeling language. The constructs most often found in metamodeling languages such as the Meta Object Facility 1.4 (MOF) [15], the Eclipse Modeling Framework EMF [9] and even the Graph eXchange Language Metaschema [23] are strikingly similar since they are all based on the object-oriented software paradigm. A metamodel is defined as a collection of classes and properties while a model is an instance of such classes and properties.

However, the recent MOF 2.0 [17] and UML 2.0 Infrastructure [18] metamodeling languages introduce new concepts, mainly: *subset* properties, *derived union* properties and property *redefinitions*. These concepts are supposed to be useful in defining a new modeling language as an extension of an existing one. Unfortunately, very little is told in [17, 18] about the actual meaning of these new constructs. This is a critical omission since these concepts are heavily used in the definition of the Unified Modeling Language 2.0 [19].

We consider that a precise definition of a metamodel is necessary in order to construct tools to edit, query and transform models. We can create such tools using a general-purpose programming language and a model repository library, for example EMF [9] or NMR [13], but also using declarative model transformation languages such as ATL [5] or QVT [16]. In any case, a precise definition of the abstract syntax of a modeling language as provided by a metamodel is necessary to define programs and transformations that operate over models based on a given language and to guarantee interoperability between these tools.

In this article, we present a set-theoretic nonmetacircular formalization of a metamodeling language that supports what we consider to be the core features of MOF 2.0 and the UML 2.0 Infrastructure, including the new subset properties. We also present the pre- and postconditions for model editing operations as well as their implementations. These basic operations are the elemental building blocks for a model repository. Although this article only presents a theoretical framework, we believe it represents an important contribution that can influence the implementation of model repositories for the UML 2.0 language.

We proceed as follows. In Section 2, we explain intuitively the core concepts of OMG modeling languages and models, and provide some examples on where the new features of MOF 2.0 can be used to define extensions to modeling languages. We present a simple formalization of the structure and constraints of metamodels and models in Section 3. In Section 4, we present pre- and postconditions for operations on models, meaning element creation and deletion as well as insertion and removal operations for both unordered and ordered slots consisting of unique elements. We acquire

requirements from the current usage of these constructs in the UML 2.0 Infrastructure as well as our understanding of how modeling frameworks should work. We also show examples of the operations. We show the implementation of the operations in Section 5 and describe the notion of different insertion strategies into ordered slots. We finally conclude in Section 6 by describing related and future work.

2 MOF 2.0 and UML 2.0 Infrastructure as Metamodeling Languages

The purpose of MOF 2.0 and the UML 2.0 Infrastructure is to define new modeling languages that can be used in software and system development. These two languages share a common core that is often named Essential MOF (EMOF). In this Section, we present the main concepts in this core language informally, focusing on new features with respect to MOF 1.x. A more thorough definition of these features will be presented in the next Section.

2.1 Classes and Properties

The main concepts used to describe the abstract syntax of a modeling language are classes and properties. A class represents a concept in a modeling language such as a UML Use Case or a Transition in a Statechart, while a property represents a feature of such a concept such as the fact that a Use Case has a name or a Transition has an event trigger.

These classes and properties can then be instantiated into a finite set of elements and slots that form a model. Each element, consisting of slots, has a class as its type and each slot has a property as its type. A slot keeps a collection of other model elements as its contents and the type of these elements and their number is constrained by its property. Elements can only occur once in the contents of a slot. Also, a property can optionally denote a composition of elements. An element can only be placed in a single composition slot at a time, called its owner. Finally, a slot can be ordered. In this case, the contents of a slot is represented by an ordered set.

The structure of a metamodel is often described visually using UML class diagrams, while models are represented using object diagrams. As an example, the left part of Figure 1 shows a metamodel for a graph. This diagram shows two classes: *Vertex* and *Edge*, and four properties: *from*, *to*, *outgoing* and *incoming*. The properties *from* and *to* belong to *Edge* and have a multiplicity constraint of 1, i.e., each element of type *Edge* should have exactly one *from Vertex* and a *to Vertex*. The properties *outgoing* and *incoming* have a multiplicity of $[0..\infty]$, i.e. a node can have any number of incoming and outgoing vertices.

Each property has another property as its opposite. Together they define an association that is represented as a single line. In the example, we have the *from-outgoing* and the *to-incoming* associations. At the model layer, this bidirectionality means that when a *Node n* has a *Vertex v* in its *outgoing* slot, the *Vertex v* will have *Node n* in its *from* slot.

We can depict models as object diagrams. In this case, each element is depicted as a rectangle, while the contents of each slot are represented as directed arcs between nodes, labeled by the name of the slot. That is, we do not represent the slots in an object diagram but their contents. An example object diagram based on our metamodel for graphs is shown in the right part of Fig. 1.

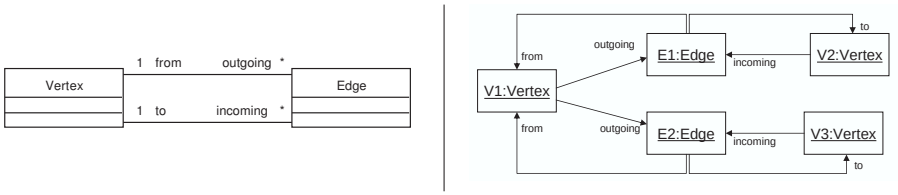


Figure 1: (Left) Metamodel for a Graph; (Right) Example Model

2.2 Language Extension

We have seen that classes and properties are the building blocks to define new modeling languages. However, we are often not interested in creating new modeling languages, but in extending an existing one.

There are two important reasons to allow the extension of a modeling language. The first one is to reduce the complexity of the definition of large languages such as UML 2.0. The current definition of UML 2.0 has been split into different packages that can be studied and implemented independently of the others. These packages are put together into a single language using the package extension mechanisms. The second reason is to allow the definition of domain specific modeling languages that share common features. For example, the UML 2.0 Infrastructure contains a basic modeling kernel with definitions that can be useful in defining profiles or other domain-specific modeling languages (DSML). This can simplify the definition of a new DSML while avoiding the so-called Tower of Babel of DSML languages [10], where the use of many unrelated modeling languages hinders the development of a new system instead of simplifying it.

Let us assume that we wish to create a metamodel for bipartite graphs based on our metamodel for general graphs. This is an example, adapted from [22], of the definition of a new language based on an existing language. An initial metamodel for bipartite graphs is shown in Figure 2. In this new language, there are two types of vertices, blue and red, and two types of edges, red-to-blue and blue-to-red.

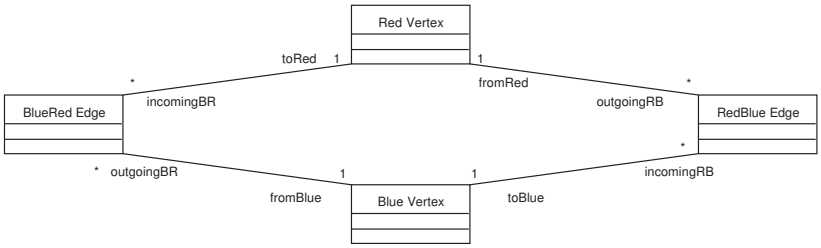


Figure 2: Metamodel for a Bipartite Graph

The metamodel for a bipartite graph as it is shown in Fig. 2 cannot represent red-to-red or blue-to-blue edges, as we intended. However, this metamodel has no relation with the metamodel for general graphs shown in Fig. 1. This means that programs and model transformations that traverse and extract information on general graphs based on the initial metamodel will not work on bipartite graphs.

Based on this discussion, we consider that an important requirement for creating new extensions to an existing modeling language is Liskov Substitutability [12]. Generally, this means that programs, queries and transformations designed for a model-

ing language should work on models based on extensions of the original modeling language. As a consequence, a language extension should not be able to arbitrarily redefine or remove classes or properties from a language.

MOF 2.0 proposes mainly three extension mechanisms for metamodels: class specialization, property subsets and unions, and property redefinitions. Class specialization is identical with class inheritance in object-oriented languages. A specialized class inherits all the properties of its base classes, and it can define new properties. Subset and union properties are a mechanism to define the relationship between the properties in a specialized class and its base classes. Finally, property redefinition allows us to arbitrarily replace a property with another one.

In the context of this paper, we require that class specialization and property subsetting should not create additional constraints on the original elements and slots instantiated from the original classes and properties. On the other hand, the designer or a language extension can introduce new constraints on the specialized classes and properties. Finally, we should not require that the original language needs to contain explicit extension points since the designer of the original language cannot tell in advance if somebody might wish to extend the language in the future.

We should note that MOF 2.0 and the UML 2.0 Infrastructure contain other concepts to define language extensions such as packages and package imports. They do not however influence the relationship between model elements. Therefore, we study only the semantics of class specialization and subset properties since we consider that these are the main novelties in MOF 2.0 and the core mechanism for language extension.

In the rest of this Section we discuss the new language extension mechanisms in more detail.

2.2.1 Class Specialization and Property Subsetting

Class specialization is represented diagrammatically as a an edge between the base class and the specialized class with a triangular arrow head pointing to the base class. Property subsets can be represented by adding a label "{ subsets }" next to a property or by connecting two associations with a specialization edge with the same label. We can see an example of these two equivalent notations in Figure 3.

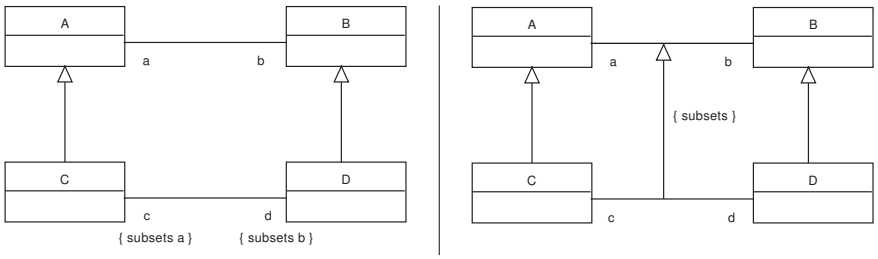


Figure 3: Two Alternative Notations for Subsetting

The intuition behind the metamodel in Fig. 3 is as follows: An element of type *C* has two slots that correspond to properties *b* and *d*. The slot representing *d* will be a subset of the slot representing *b*. Elements of type *B* can be inserted into slot *b* and elements of type *D* can be inserted into slots *b* and *d*. At any moment, the contents of the slot *d* should be a subset of the contents of the slot *b*.

Class specialization works so that an instance of a class has slots according to the properties of that class or any of its transitive superclasses.

We can use specialization and subset properties to create a new metamodel for a bipartite graph for our running example. The classes *Blue Vertex* and *Red Vertex* will now be specializations of *Vertex*. Also, the four *fromRed* and *toBlue* properties will become subsets of the *from* and *to* properties, and similarly for *incomingBR*, *incomingRB*, *outgoingBR* and *outgoingRB*. The resulting metamodel is show in Figure 4 while a model based on this metamodel is shown in Figure 5. The benefit is that e.g. graph traversal algorithms which worked on the initial metamodel in Fig. 1 should still work for bipartite graphs when using the metamodel in Fig. 4.

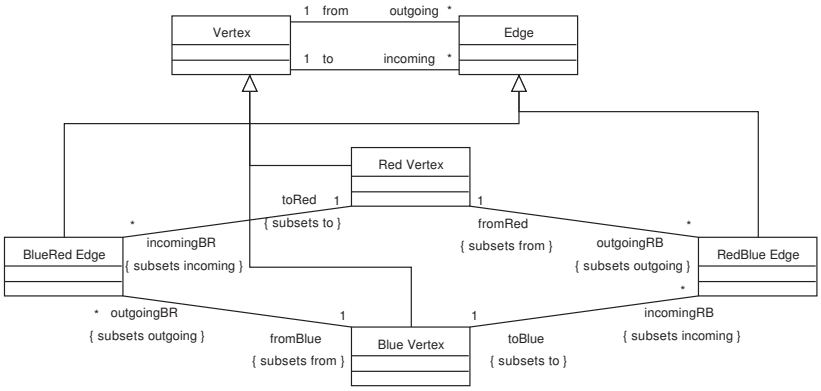


Figure 4: Metamodel for a Bipartite Graph as an Extension of the Metamodel for a General Graph

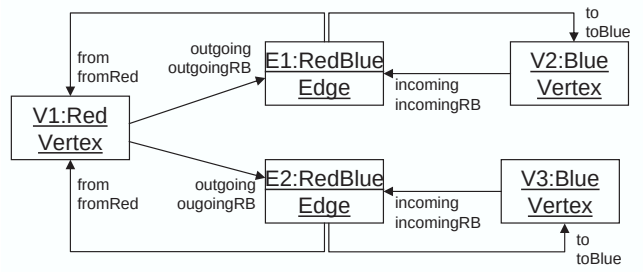


Figure 5: Example Model for the Graph Metamodel

2.2.2 Union Properties and Derived Unions

The last extension mechanism presented in the MOF 2.0 that we will discuss in this paper is union properties. In our terminology, if a property has properties that subset it, it is a union property. It is not necessary to declare a property as a union, since a designer of a metamodel cannot know in advance if a new subset property will be defined in the future.

The UML 2.0 Infrastructure also introduced the concept of derived union. The standard states on page 126 that *"This means that the collection of values denoted by the property in some context is derived by being the strict union of all of the values denoted, in the same context, by properties defined to subset it. If the property has a multiplicity upper bound of 1, then this means that the values of all the subsets must*

be null or the same." In other words, a derived union property can be seen as the strict union of its subsets. A slot with a property that is a derived union cannot contain elements that do not appear in any of its subsets.

Another way to define derived content is to create an arbitrary query operation, This has been done in the Eclipse Modeling Framework using the so called volatile attributes as explained in [6]. This way, the contents of a slot are defined by evaluating the associated query. The drawback is that there is no strict mathematical relationship between the derived property and any other properties. The benefit is that it does not restrict the metamodel creator in any way.

2.3 Language Extension Issues

There are some issues with language extension that immediately need to be taken into account.

2.3.1 Multiple Generalizations

We should note that a metamodeling language should support multiple inheritance since it is used extensively in MOF, as has been noticed by e.g. Anneke Kleppe [11]. Multiple inheritance forms very complicated inheritance hierarchies, among them the *diamond inheritance* structure. This leads to a possibility where property subsetting also has a diamond (or even more complicated) structure. In Figure 6 an element of type D has four slots: a', b', c' and d', where d' is a subset of b' and c', while b' and c' are subsets of a'. As we shall see, this complicates the definitions of subsetting, especially in subsetting of ordered properties.

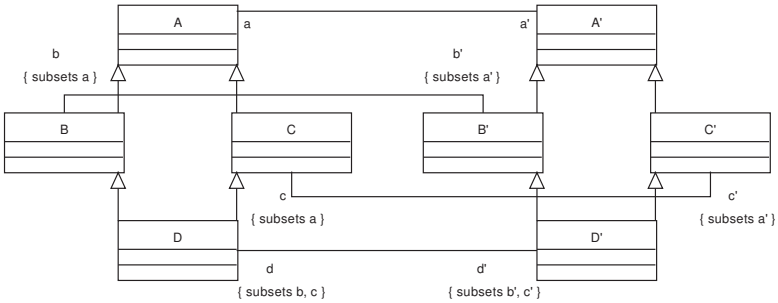


Figure 6: Diamond Inheritance and Diamond Subsetting

2.3.2 Subset and Union Properties in the same Class

Subset properties can be useful even when they are not used in combination with class specialization. That is, we can define a property and its subsets in the same class.

As an example, we can create a simplified metamodel for UML class diagrams. The metamodel is show in Figure 7 and it is inspired by the UML 2.0 metamodel (e.g. Figure 50 of [18]). We first provide the general concept of a container and its children elements using the *Container* and *Element* classes. Each *Container* element has a slot named *ownedElement* representing its contents.

Then we specialize *Container* into a *Class* and add two subset properties called *ownedAttribute* and *ownedOperation* to keep attributes and operations. These properties are a subset of the *ownedElement* property. We also add two subsets of *owne-*

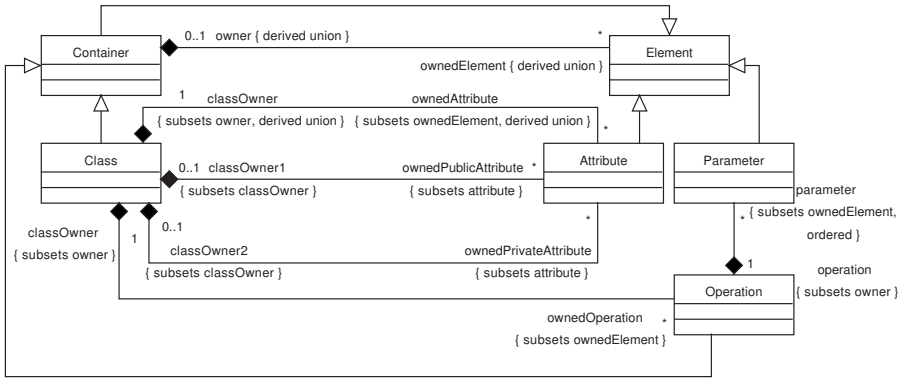


Figure 7: Example Generic Container

dAttribute to *Class* called *ownedPublicAttribute* and *ownedPrivateAttribute*. This is an example of how different subset properties may refer to the same class.

In this metamodel, *Attribute* and *Operation* elements should always be owned by a *Class* since their *classOwner* property has a multiplicity of exactly 1. However, an *Attribute* can be public or private depending on in which slot it is placed.

This notion of general containment is used extensively in UML 2.0, spanning several inheritance relationships. The language uses the qualifier "{ union }" whereas we use "{ derived union }" to mean that the contents of both owner, *ownedElement*, *classOwner* and *ownedAttribute* slots are derived (read-only) from the contents of the subsetting slots.

This metamodel also shows an ordered property: since the order of the *Parameter* elements in an *Operation* is relevant in this model, the *ownedParameter* property is ordered. This is an interesting observation since the superset *ownedElement* is an unordered property.

An example of a class model is show in Figure 8.

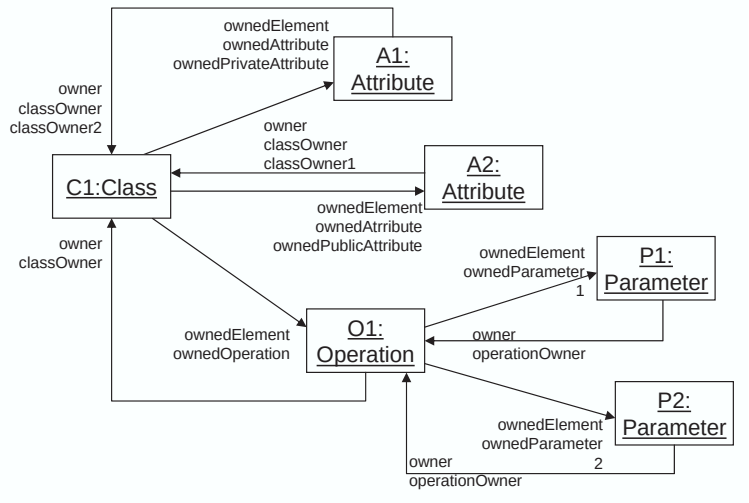


Figure 8: Example of a Class Model Using the Extended Metamodel Approach

2.3.3 Property Subsetting Follows Class Specialization

Considering Figure 3 further, we investigate two fundamental cases in Figure 9. The first one depicted on the left side of the Figure questions the validity of the scenario where class *C* does not subclass *A* even though it has a property which subsets another property from *A*.

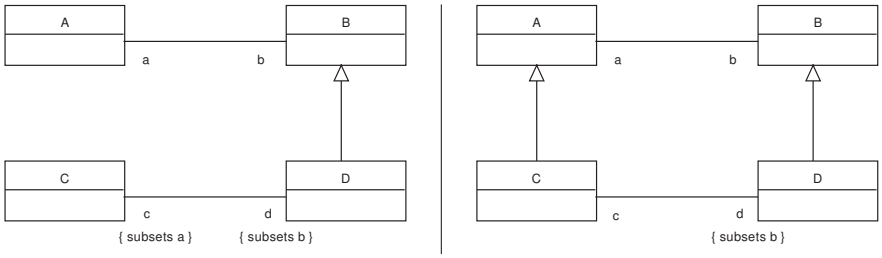


Figure 9: (Left) Subsetting without Generalization; (Right) Subsetting Only One Property of an Association

Let us take an element e_c of type *C* and an element e_d of type *D*. Modifying the *d* slot of e_c, i.e, modifying $e_c.d$ (by inserting or removing e_d), should modify $e_c.b$ as well, due to subsetting. But the slot *b* does not exist in e_c. This is a contradiction and the conclusion must thus be that property subsetting "follows" class generalization. In our example, either class *C* needs to be a subclass of class *A* or the subsetting has to be removed.

The same remark is also stated in the UML 2.0 Infrastructure [18] on page 126: *A property may be marked as a subset of another, as long as every element in the context of the subsetting property conforms to the corresponding element in the context of the subsetted property.*

2.3.4 The Opposite of a Subset Property Should be a Subset

Finally, let us consider the metamodel on the right side of Figure 9. In this metamodel, the subset property *c* has an opposite property *a* that it is not a subset.

It can easily be seen that this idea is not sound. Let us take an element e_c of type *C* and an element e_d of type *D*. Modifying $e_c.d$ necessarily modifies $e_d.c$ as well due to bidirectionality. Due to subsetting between properties *b* and *d*, $e_c.b$ is also modified—without yet going into details about the exact semantics, since we have only discussed the intuition behind the model layer so far. Then due to bidirectionality, $e_d.a$ is also modified and the final effect is as if *c* would subset *a*. The conclusion is that we claim that *c* indeed needs to subset *a*, if for nothing else than documentation purposes.

There are several faults in the metamodels for MOF 2.0 and UML 2.0 where this rule is violated. Fortunately, the correction is simple by saying that (in our example) *c* needs to subset *a*.

2.4 Alternative Language Extension Mechanisms: Covariant Specialization of Properties

There exists at least one other approach to language extension, covariant specialization. As an example, take an element e_c of type *C* in Figure 10. In a covariant environment, it is not possible to insert elements of type *B* into the *b* slot of an element, only elements of type *D* into the *d* slot. The *c*-*d* association is a *covariant specialization* of the *a*-*b*

association and classes C and D are not subtypes but covariant specializations of classes A and B, respectively. The a-b association has been rendered obsolete in the context of element e_c.

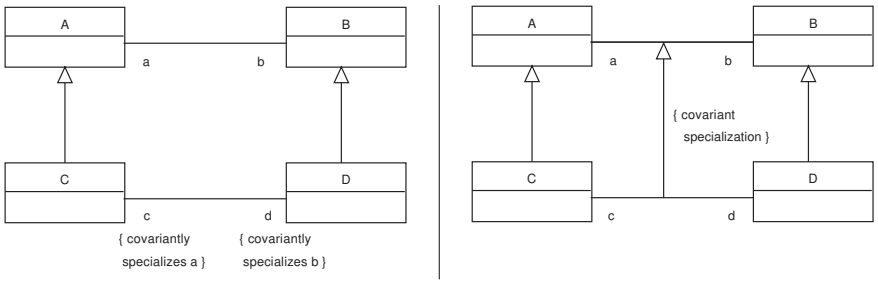


Figure 10: Two Alternative Notations for Covariant Specialization

In object-oriented programming, function parameter type contravariance and return type covariance are rather inconvenient in practical situations [7] and thus a type-unsafe function parameter type covariance is used for specialization. A similar argument also holds for element slots and not only object methods. Property subsetting aims to provide a new way to represent relationships between elements. It must nevertheless be noted that as Giuseppe Castagna has asserted [7], there are uses for a covariant environment when compared with a contravariant or invariant environment. Thus subsetting and covariance are not opposing but complementing constructs in modeling and thus object-oriented programming.

The major difference between covariance and subsetting is that in a covariant environment, substituting an element of a specific type with an element which is a covariant specialization of that type can result in programs no longer working. Subsetting allows the slots defined by the properties in a superclass to be used in an instance of a subclass.

In this Section we have seen that classes and properties are the basic building blocks to create new metamodels, while class specialization and property subsetting are the basic mechanisms to define metamodel extensions. In the rest of the paper, we will define these concepts formally and provide the basic definitions and algorithms needed to create and edit models according to a given metamodel.

3 A Simple Metamodeling Language

In this Section we present the definition of a simple metamodeling language that contains the core concepts of MOF 2.0 and the UML 2.0 Infrastructure. We ignore classes that represent primitive datatypes such as integers, strings and enumeration values without loss of generality.

One of the main characteristics of these languages is that they are not inspired by previous research in formal languages but by the object-oriented software development paradigm. As such, a modeling language is interpreted as a collection of classes while a model is a collection of instances from these classes. Although this is a valid metaphor, it should be developed thoroughly so a metamodel can serve as a grammar: a formal definition of the syntax of a language. The definition of a metamodeling language should include a procedure for the language membership problem, determining if a given model belongs to a given modeling language, and to be able to enumerate all models in a given language.

Our modeling framework acts as our metamodeling language. We represent all metamodels as a tuple $MM = (C, P, \text{generalizations}, \text{properties}, \text{characteristics})$, where C is a set of classes, P a set of properties and $C \cap P = \emptyset$. We define the generalizations of a class with the function $\text{generalizations} : C \to \mathcal{P}(C)$. The direct generalization relation is defined as $g \stackrel{\text{def}}{=} \{(c_1, c_2) \mid c_2 \in \text{generalizations}(c_1)\}$. We require the directed graph representing the class generalization relation (C, g) to be acyclic. Also, we denote by $\subseteq_c$ the extended generalization between classes that is defined as the reflexive transitive closure of the generalization relation: $\subseteq_c \stackrel{\text{def}}{=} g^*$. We can prove that $\subseteq_c$ is a partial order since it is reflexive and transitive by definition and antisymmetric due to the fact that the generalization graph is acyclic.

Which properties belong to each class is given by the function $\text{properties} : C \to \mathcal{P}(P)$. Every value of the function properties is a disjoint subset of P. Thus, we can define $\text{owner} : P \to C$ which denotes the unique owner c of a property p where $p \in \text{properties}(c)$. The effective properties of a class are those defined by the class itself and transitively by any of its generalizations.

Finally, the characteristics of a property represent constraints in the elements that can be place in the slots. In this paper, we define

$\text{characteristics} \stackrel{\text{def}}{=} (\text{lower}, \text{upper}, \text{opposite}, \text{ordered}, \text{composite}, \text{derived}, \text{supersets})$

as a tuple of functions detailing the properties further:

- $\text{lower} : P \to \mathbb{Z}^{0+} \setminus \infty$ represents the lower multiplicity constraint of a property (0, 1, 2, ...)
- $\text{upper} : P \to \mathbb{Z}^{+}$ represents the upper multiplicity constraint (1, 2, ..., ∞).
- $\text{opposite} : P \to P$ is a bijective function that yields the opposite of a property. The opposite of a property cannot be itself: $(\forall p \in P \mid \text{opposite}(p) \neq p)$. A property is the opposite of its opposite: $(\forall p \in P \mid \text{opposite}(\text{opposite}(p)) = p)$.
- $\text{ordered} : P \to \mathbb{B}$ is true if a property is ordered.
- $\text{composite} : P \to \mathbb{B}$ is true if a property is composite.
- $\text{derived} : P \to \mathbb{B}$ is true if a property is derived.
- $\text{supersets} : P \to \mathcal{P}(P)$ represents the set of properties of which a property is a subset. The graph representing the property superset relation $(P, \{(p1, p2) \mid p_2 \in \text{supersets}(p_1)\})$ must be acyclic.

For convenience, we define the function $\text{subsets} : P \to \mathcal{P}(P)$ as the inverse of supersets. We denote subsetting between properties by the $\subseteq_p$ relation, i.e. $\subseteq_p \stackrel{\text{def}}{=} \{(p, q) \mid q \in \text{supersets}(p)\}^*$. We can write $\subseteq$ instead of $\subseteq_p$ or $\subseteq_c$ without ambiguity, since one is a relation over classes whereas the other is a relation over properties. We also define $a \subset b \stackrel{\text{def}}{=} a \subseteq b \wedge a \neq b$ for both classes and properties.

Finally, we denote by $s \prec t$ that s is a direct subset of t, i.e., $s \prec t \stackrel{\text{def}}{=} s \subset t \wedge \neg(\exists u \mid s \subset u \subset t)$. The expression $s \mid\mid t$ means $\neg(s \subseteq t) \wedge \neg(t \subseteq s)$, i.e., there is no order defined between these properties.

Based on the discussion in the previous Section, we should consider three additional constraints over the structure of a metamodel. First, a property can subset another property only from the transitive superclass closure of its owner: $(\forall p, q \in P \mid p \subseteq_p q \Rightarrow \text{owner}(p) \subseteq_c \text{owner}(q))$. Also, the opposite of a subset property should be a subset: $(\forall p, q \in P \mid p \subseteq_p q \Rightarrow \text{opposite}(p) \subseteq_p \text{opposite}(q))$. Finally, despite Figure 7, we require in our formalization that a property has the same ordering characteristic as its subsets $(\forall p, q \in P \mid p \subseteq_p q \Rightarrow \text{ordered}(p) = \text{ordered}(q))$. Given these constraints it can be shown, following a similar argument as for $\subseteq_c$, that $\subseteq_p$ is a partial order.

We can now define a metamodel simply as any nonempty finite subset of the set of classes C, $MM_x \subseteq C$. Note that generalization of classes and the opposite of a property can be defined across several metamodels.

3.1 Models

We define the infinite set of all models as $\mathcal{M} = \{M \mid M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})\}$. M comprises all the models in a system at some specific time. E is a finite set of elements and S is a finite set of slots. Each element in E has a type defined by a class in a metamodel, $\text{type} : E \to C$ and a set of slots defined by the function $\text{slots} : E \to \mathcal{P}(S)$. Every value of the function slots is a disjoint subset of S. Thus, we can define $\text{owner} : S \to E$ which denotes the unique owner e of a slot s where $s \in \text{slots}(e)$. Each slot corresponds to a property as defined by the function $\text{property} : S \to P$. Slots consist of element references. The function $\text{elements} : S \to (E, \prec)$ returns a total ordered set of elements of its argument slot s if $\text{ordered}(\text{property}(s))$ is true, otherwise $\text{elements} : S \to \mathcal{P}(E)$ returns an unordered set of elements. The slot thus describes the connection from its owner element to the elements in the slot. There is no actual ordering defined between the elements in an ordered slot; they merely have an assigned position in s. We stress that no matter whether a slot is ordered or not, an element can never occur twice in it.

For an element e, we can also access its slot corresponding to property p using the notation $e.p$, as is common in many object-oriented languages. Because each property has an opposite property, the elements in a slot owned by e will each have a slot which contains e. This feature is called bidirectionality and must never be violated. In other words, if an element e has a slot s which contains e', then e' must also have a specific slot s' which contains e.

For convenience, we define the size of a slot to be the amount of elements in that slot: $(\forall s \in S \mid \#s \stackrel{\text{def}}{=} \#\text{elements}(s))$. For the elements of an ordered set, we say $s[i]$ to denote the element at the zero-based index i in the ordered set s. We define the function $\text{parent} : E \to \mathcal{P}(E)$ to return a set consisting of the parent element of the argument, if any, otherwise the empty set:

$$\text{parent}(e) \stackrel{\text{def}}{=} \{x \mid x \in E \wedge (\exists s \in S \mid s \in \text{slots}(x) \wedge \text{composite}(\text{property}(s)) \wedge e \in \text{elements}(s))\}$$

We define the slot subsetting relation as $\subseteq_s \stackrel{\text{def}}{=} \{(s,t) \mid \text{property}(s) \subseteq_p \text{property}(t)\}^*$. It can be split into several partial orders, one for each slot and its super- and subsets.

The contents of a slot s subsetting another slot t must be a subset of the contents of t. Also, MOF [17] tells us on page 59 that *"The slot's values are a subset of those for each slot it subsets."* For ordered slots, we also wish to preserve order, i.e., when elements occur in a specific order in s, they should occur in the same order in t, although t might contain more elements in between. We denote $a \prec_x b$ if element a precedes element b in a specific ordered slot x. A slot s (transitively) subsetting another slot t is denoted by $s \subset_s t$. Again, we can drop the subscript without ambiguity.

The following list contains all constraints that models should fulfill with respect to their metamodel. Among the relevant constraints for models are that there can only be one owner element for each slot. An element may be in at most one composite slot, and that composition is acyclic. A slot can only contain elements which are a subclass of the owner of the opposite of a property, i.e., slots in our framework are strongly typed. These constraints also serve as an invariant which must be maintained by any operation on models.

- Valid slots in element (1):
  $(\forall e \in E \mid (\forall s \in S \mid s \in \text{slots}(e) \Rightarrow \text{type}(e) \subseteq \text{owner}(\text{property}(s))))$

- Valid slots in element (2):
  $(\forall e \in E \mid (\forall c \in C \mid \text{type}(e) \subseteq c$
  $\Rightarrow (\forall p \in \text{properties}(c) \mid (\exists! s \in \text{slots}(e) \mid \text{property}(s) = p))))$

- Slot bidirectionality: $(\forall s \in S \mid (\forall e' \in \text{elements}(s) \mid (\exists! s' \in S \mid \text{owner}(s') = e' \wedge$
  $\text{opposite}(\text{property}(s')) = \text{property}(s) \wedge \text{owner}(s) \in \text{elements}(s'))))$

- Multiplicity: $(\forall s \in S \mid \text{lower}(\text{property}(s)) \leq \#s \wedge \#s \leq \text{upper}(\text{property}(s)))$

- At most one parent : $(\forall e \mid \neg(\exists s_1, s_2 \mid s_1 \neq s_2 \wedge \text{composite}(\text{property}(s_1))$
  $\wedge \text{composite}(\text{property}(s_2)) \wedge e \in \text{elements}(s_1) \wedge e \in \text{elements}(s_2))$

- Acyclic composition: $(\forall e_1, \ldots, e_n, e_{n+1} \in E, \exists s_1, \ldots, s_n \in S \mid (\forall i \mid 1 \leq i \leq n$
  $\Rightarrow \text{owner}(s_i) = e_i \wedge \text{composite}(\text{property}(s_i)) \wedge e_{i+1} \in \text{elements}(s_i))$
  $\Rightarrow e_1 \neq e_{n+1}) \quad (n > 0)$

- Type correctness: $(\forall s \in S \mid (\forall e \in \text{elements}(s)$
  $\mid \text{type}(e) \subseteq \text{owner}(\text{opposite}(\text{property}(s)))))$

- Unordered slots: $(\forall r, s \in S \mid r \subseteq s \Rightarrow \text{elements}(r) \subseteq \text{elements}(s))$

- Ordered slots: $(\forall x, y \in E, r, s \in S \mid x \in \text{elements}(r) \wedge y \in \text{elements}(r) \wedge x \prec_r y \wedge r \subseteq$
  $s \Rightarrow x \in \text{elements}(s) \wedge y \in \text{elements}(s) \wedge x \prec_s y)$

The last two constraints are specific to unordered and ordered slots with respect to property subsetting. We call these two constraints the *inherent subsetting rules*, or ISR.

We can now define a valid model M_x as a set of elements, $M_x \subseteq E$. We note that our framework supports connections from elements in one model to elements in another model. As such, the concept of "one model" is not too important; we would rather stress the importance of "all models", i.e., M. In our framework, it is up to the user to define at any time what elements comprise one model.

3.2 Metamodel Membership Function

Given a model $M_x \subseteq E$ we should be able to answer whether it is an instance of a given metamodel $MM_x \subseteq C$. This question can be asked in two slightly different ways. First, we can ask whether M_x is an instance of MM_x and not of an extension of MM_x. This is done by requiring the base type of all elements to be in the class definitions of MM_x:

$(\forall e \in M_x \mid \text{type}(e) \in MM_x)$

On the other hand, we can also allow M_x to be an instance of either MM_x or an extension of it. This definition takes metamodel extensibility into account.

$(\forall e \in M_x \mid (\exists c \in MM_x \mid \text{type}(e) \subseteq c))$

We should remark that in addition to either one of the above two propositions, all model constraints must also hold. An empty model consisting of no elements is considered by definition to be a member of any metamodel.

3.3 Inconsistent Metamodels

We have seen that different property characteristics such as multiplicity, composition or subsetting allows us to define rich and extensible metamodels. However, we should consider whether there are combinations of these characteristics that define inconsistent metamodels, i.e., metamodels that do not represent any model, except for the trivial empty model. To avoid this situation, we may require to define additional constraints in a metamodel.

3.3.1 Multiplicities

Since the number of elements in a slot is bounded by the multiplicity characteristics of its property, we require that the lower value in a multiplicity range should be less than the upper value: $(\forall p \in P \mid \text{lower}(p) \leq \text{upper}(p))$. Otherwise, the metamodel is inconsistent.

Also, it can easily be seen that a property subsetting another property must have a lower (or the same) upper limit than the other property. This can be formalized with

$(\forall p \in P \mid (\forall q \in \text{supersets}(p) \mid \text{upper}(p) \leq \text{upper}(q)))$

The justification for this constraint can be seen with slots s and r such that $s \subset r$, $\text{property}(s) = p$, $\text{property}(r) = q$, $\text{upper}(p) > \text{upper}(q)$ and by filling the slot s with elements so that $\#s = \text{upper}(p)$. Then $\#r \geq \#s = \text{upper}(p) > \text{upper}(q) \Rightarrow \#r > \text{upper}(q)$, which violates the upper limit of q.

There are no restrictions on the lower limits of the properties, since more elements can always be inserted into a slot until its size is at least that of the lowest limit in any transitive sub- or superset.

3.3.2 Composition

In Figure 11, we see different cases with composite and noncomposite properties. Cases (1) and (2) are quite self-explanatory. Case (3) can be considered legal by discounting the composition at the *C*-*D* association without any loss in information, since any elements owned via the *C*-*D* association must also be owned via the *A*-*B* association. Case (4) is illegal since any elements of types *C* and *D* that are connected at the *C*-*D* association are also connected at the *A*-*B* association, thereby creating a cyclic composition and violating a model constraint. Thereby the following metamodel constraint must be added:

$(\forall p \in P \mid \text{composite}(p) \Rightarrow \neg(\exists q \in P \mid p \subset q \wedge \text{composite}(\text{opposite}(q)))$

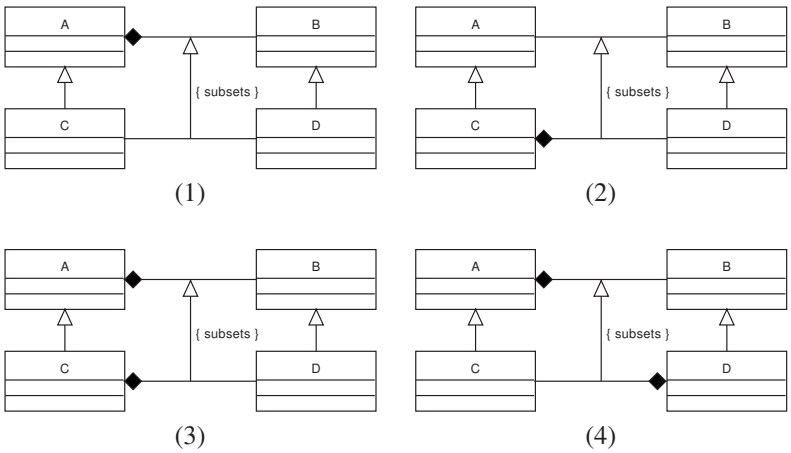


Figure 11: Subsetting with Composite and Noncomposite Properties

We can find examples of the three first cases in the UML 2.0 metamodel, all in Figure 73 of [18]. Case (1) can be found in the *association-memberEnd* association, case (2) in the *owningAssociation-ownedEnd* association and case (3) in the *class-ownedAttribute* association.

In this Section we have presented basic definitions of metamodel and models, including the definitions of the metamodel and model constraints. We have also presented

a membership function for models. In the following Section, we will define the basic operations to edit models.

4 Basic Edit Operations for Models

In this Section we present the basic operations to create and remove elements from models as well as to insert or remove an element from a slot. These four operations are the basic edit operations for models that are necessary to implement a model repository and a model transformation system.

We define these operations using a pre- and postcondition specification. We first describe element creation and deletion. Then, we describe the case of insertion into ordered or unordered slots and finally the case of removing elements from slots. Actual implementations of these operations are described separately in Section 5. The pre- and postconditions are described as separate enumerated clauses. All of the clauses in the precondition must hold for the operation to succeed, and all the clauses of the postcondition must be guaranteed by an implementation.

We should note that the basic edit operations can invalidate a slot with respect to the multiplicity constraints. As an example, assume a class c has a property p such that $\text{lower}(p) > 0$. Creating an instance of c would need a slot s such that $\text{property}(s) = p$ and $\#s > 0$, i.e., s needs to have at least one (possibly new) element to be valid. That is, one single basic edit operation does not guarantee that the resulting slot owned by c is valid. As a consequence, a model transformation must be defined as a composition of these basic operations.

For succinctness and understandability of presentation, we only describe the semantics of an operation in the context of a slot and its super- and subsets. This approach is valid because property subsets always come in pairs of two isomorphic partial orders, one for each side of the associations, as can be seen in e.g. Figure 6.

Thus, in the context of an insertion into or removal from a slot, that slot is part of a partial order. Each slot in that partial order has an opposite slot in that context, which all together form another partial order. When modifying a slot, similar actions must be taken for the slots in the opposite partial order for bidirectionality to hold. This means that the actual operations must, where necessary, be augmented with an additional index parameter for the ordered slots in the opposite partial order.

The context in which the pre- and postconditions are evaluated is the model data, $M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements})$. In postconditions, the new values of variables are denoted with tick marks, otherwise the old values before the execution is assumed. Thus, $M' = (E', \text{type}', \text{slots}', S', \text{property}', \text{elements}')$ refers to new values in the model data.

4.1 Element Creation

The operation create : $\mathcal{M} \times C \to \mathcal{M} \times E$ creates a new element of type $c \in C$ and has no preconditions. It will also be a root element, i.e., it will not have any parent. The postcondition is that there must be exactly one new element in the set of elements:

1. $(\exists! e \in E' \mid E' \setminus \{e\} = E \wedge \text{type}(e) = c)$

4.2 Element Deletion

The operation delete : $\mathcal{M} \times E \to \mathcal{M}$ deletes an element. We require the element being deleted to have no connections to other elements via its slots. Therefore the precondition for deleting an element e is:

1. $(\forall s \in \text{slots}(e) \mid \#s = 0)$

The postcondition is that the element must no longer be in the set of elements:

1. $E' = E \cap \{e\}$

4.3 Element Insertion into an Unordered Slot

Consider an operation insert : $\mathcal{M} \times S \times E \rightarrow \mathcal{M}$ such that insert(M,s,e) inserts element e into slot s. The intuition behind our insertion operation is that all supersets of s must contain the new element e for the model constraints, especially the ISR, to hold. The clauses for the precondition for element insertion into an unordered slot are thus:

1. $\neg$derived(property(s))
2. $\neg$ordered(property(s))
3. $e \notin$ elements(s).
4. type$(e) \subseteq$ owner(opposite(property(s)))
5. $(\exists t \in S \mid s \subseteq t \wedge \text{composite}(\text{property}(t)) \wedge e \notin \text{elements}(t)) \Rightarrow \text{parent}(e) = \emptyset$

The clauses state that (1) we are not modifying a read-only slot, (2) the slot is unordered, (3) the element must not yet exist in the slot, (4) that we obey the rules of strong typing and (5) we do not create a connection to a second parent for e.

The postcondition for element insertion is simple. We wish element e to be found in the slot s and all its transitive supersets. All the model constraints except for the multiplicity constraints must also hold as a postcondition.

1. $(\forall t \in S \mid s \subseteq t \Rightarrow \text{elements}(t') = \text{elements}(t) \cup \{e\})$ (Note $s \subseteq s$)

We can depict partially ordered sets using Hasse diagrams. In this article we represent slots as nodes and edges between slots denote subsetting. A slot visually higher up is subsetted by the (connected) slots below it in the diagram.

An example of element insertion into an unordered slot can be seen in Figure 12. In case (1) of the Figure, we have a partially ordered set of unordered slots. Suppose we insert an element c into slot q. This triggers an insertion of c into slots p and r as well, to maintain the ISR, with the end result shown in case (2). After this, inserting c into slot t also inserts it into slot s, again to maintain the ISR, resulting in case (3). Slots p, q and r are not modified because c already existed in those slots.

It can be noted that in our semantics, an insertion into a slot never modifies any subset of that slot.

4.4 Element Insertion into an Ordered Slot

Subsetting with ordered slots is more complicated than with unordered slots, due to the need to maintain an order between the elements in different slots. We define the operation insert : $\mathcal{M} \times S \times E \times \mathbb{Z}^{0+} \rightarrow \mathcal{M}$ such that insert(M,s,e,i) inserts an element e into a slot s at index i. The precondition is otherwise identical to the case when inserting into an unordered slot, except for the check for an ordered slot. and that there exists an extra clause which calculates if the insertion into the slot and its transitive supersets is at all possible without violating the ISR.

1. $\neg$derived(property(s))
2. ordered(property(s))

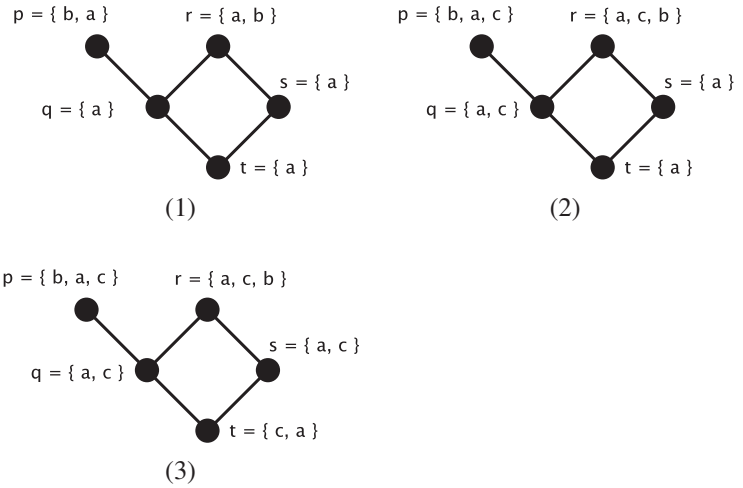


Figure 12: Example of Inserting an Element into Unordered Slots

3. $e \notin \text{elements}(s)$
4. $\text{type}(e) \subseteq \text{owner}(\text{opposite}(\text{property}(s)))$
5. $(\exists t \in S \mid s \subseteq t \wedge \text{composite}(\text{property}(t)) \wedge e \notin \text{elements}(t)) \Rightarrow \text{parent}(e) = \emptyset$
6. $\text{indices\_ok}(\{t \mid s \subset t\},$
   $\{s \mapsto [i..i]\}$
   $\cup \{t \mapsto [\text{lower\_index}(\text{index}(e,u),t,u)..\text{lower\_index}(\text{index}(e,u),t,u)] \mid s \subset t \wedge$
   $(\exists u \mid t \subseteq u \wedge e \in \text{elements}(u))\}$
   $\cup \{t \mapsto [0, \#t\ ] \mid s \subset t \wedge \neg(\exists u \mid t \subseteq u \wedge e \in \text{elements}(u))\}$
   $)$

We assume there is a function $\text{index} : E \times S \to \mathbb{Z}^{0+}$ which returns the zero-based index of an element in the contents of an ordered slot. A function $\text{lower\_index} : \mathbb{Z}^{0+} \times S \times S \to \mathbb{Z}^{0+}$ is such that $\text{lower\_index}(i,x,y)$ returns the index in x where $y[i]$ should be inserted to maintain the subset $x \subseteq y$. It is shown in Figure 13 and is used to calculate which restrictions from supersets apply to subsets when inserting an element. As an example, consider what the restriction given by element c (at index position 2) in the superset $[a,b,c,d]$ is to its subset $[a,d]$. Then $\text{lower\_index}(2,[a,d],[a,b,c,d])$ returns 1 since c should be inserted between a and d. A function $\text{lift\_interval} : S \times S \times R \to$

$\text{lower\_index}(i,x,y) :=$
  if $y[i] \in x$ then return $\text{index}(y[i],x)$
  do
    if $y[i] \in x$ then return $\text{index}(y[i],x)+1$
    else if $i = 0$ then return 0
    else $i := i-1$
  od

Figure 13: The lower_index Function

R, where R denotes integer intervals is such that $\text{lift\_interval}(x,y,[v..w])$ "lifts" the interval $[v..w]$ from x as superimposed on y (when $x \subseteq y$). It is shown in Figure 14 and

is used to calculate which restrictions from subsets apply to supersets and works as the dual of lower_index. As an example, consider the ordered sets $x = [c]$ and $y = [b,c]$. If we were to insert element a at index 0 in x, the corresponding interval for x would be $[0..0]$. This interval is superimposed onto y as the interval $[0..1]$, meaning that the same element can be inserted either before or after b in y without violating the ISR. Thus, lift_interval$(x,y,[0..0]) = [0..1]$.

$$
\begin{array}{l}
\text{lift\_interval}(x,y,[v..w]) := \\
\quad \text{if } v > 0 \text{ then } v' := \text{index}(x[v-1],y) + 1 \text{ else } v' := 0 \\
\quad \text{if } w = \#x \text{ then } w' := \#y \text{ else } w' := \text{index}(x[w],y) \\
\quad \text{return } [v'..w']
\end{array}
$$

Figure 14: The lift_interval Function

The function indices_ok : $\mathcal{P}(S) \times (S \to R) \to \mathbb{B}$ returns true if when executing indices_ok(T,F) there is a possible way to insert an element into every slot in T such that the constraints in F are satisfied. Here, $F : S \to R$ is a map from slots to integer intervals $[v..w]$ such that $v \leq w$ where e can be inserted. The function is shown in Figure 15. Here, dom F returns the domain of function F. Using the lift_interval and lower_index functions we restrict the possible intervals where e can be inserted into the slots. The intuition behind the last clause in the precondition and the definition of

$$
\begin{array}{l}
\text{indices\_ok}(\emptyset,F) := (\forall t \in \text{dom } F \mid F(t) \neq \emptyset) \\
\\
\text{indices\_ok}(T,F) := \\
\quad (\exists t \in T \mid (\forall u \in T \mid t \not\supset u) \\
\qquad \wedge R \stackrel{\text{def}}{=} \cap\{\text{lift\_interval}(c,t,[v..w]) \mid (\forall c \mid s \subseteq c \prec t \wedge F(c) = [v..w])\} \\
\qquad \mid \text{indices\_ok}(T \setminus \{t\}, F[t \mapsto R \cap F(t)]))
\end{array}
$$

Figure 15: The indices_ok Function

the indices_ok function is that we calculate the range restrictions of e which exist in any super- or subsets onto the other slots. The F function is initially created by describing constraints from supersets. F is created from three different clauses. The first, $s \mapsto [i..i]$, constrains e to be inserted at exactly index i. The second does similarly for supersets which have a superset that already has e, whereas the third initially allows all indices to be candidates for insertion. This initialization makes sure that F is restricted by the the elements e that already exist in any supersets of s. Note that any slot o such that $o \subset t \wedge s \subset t \wedge o \mid\mid s$ is outside of the transitive superset closure of s and any restrictions from it will already be visible in t and o can thus be left out from F.

Then, indices_ok calculates the constraints from subsets and does set intersection to calculate whether an insertion is possible. The actual function takes all supersets T and picks one $t \in T$ which is a bottom element, which must exist since the slots in T are part of a partial order. It then imposes all intervals from subset slots c (such that $s \subseteq c \prec t$) onto t, also including the initial constraint on t. It then recurses with a modified F until T is empty. The notation for a modified function is $f[x \mapsto y]$ which returns a new function f' such that $(\forall z \neq x \mid f'(z) = f(z))$ and $f'(x) = y$.

We claim—without proof—that if the final mapping F contains only nonempty intervals, it is possible to successfully insert e into s at index i. The postcondition is:

1. $\text{elements}'(s')[i] = e$
2. $(\forall t \in S \mid s \subseteq t \wedge e \notin \text{elements}(t) \Rightarrow t' \setminus \{e\} = t \wedge e \in \text{elements}'(t'))$

The current definitions do not tell us the exact index where to insert e into any superslot of s, only that a combination of indices exists; an index i_t for a superslot t of s must exist somewhere in the range given by $F(t)$.

An example of element insertion can be seen in Figure 16. Case (1) is the initial configuration of the slots w, x, y and z. Let us assume an insertion of element c into slot w at index position 0 occurs. The returned slot ranges where c should be inserted raises the possibilities in cases (2) to (5), depending on whether c is inserted onto the left or right side of either a in slot y or b in slot z. Cases (2) to (4) are correct solutions and our postcondition does not prefer any particular one over the another. Case (5) is not legal, because slot x cannot maintain the superset relationship as enforced by both slots y and z, as element c should occur both before a and after b in the ordered set. It is up to the implementation to choose one of the correct solutions, perhaps with guidance from the user.

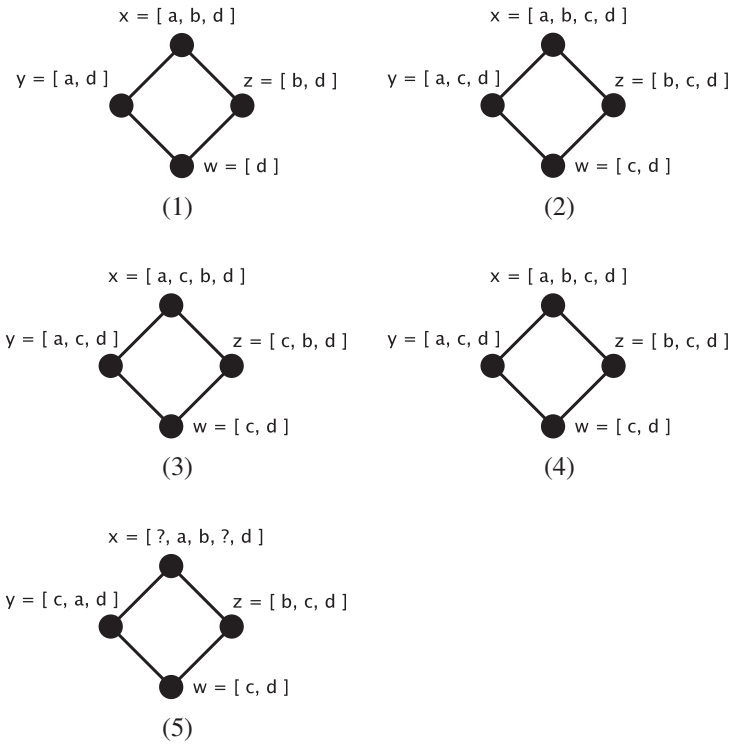


Figure 16: Example of Inserting an Element into Ordered Slots

4.5 Element Removal from a Slot

The operation remove : $\mathcal{M} \times S \times E \rightarrow \mathcal{M}$ is defined such that remove(M, s, e) removes the element e from s and all its subsets, as well as from those supersets which would not acquire e via some other subset which is not comparable to s. Element removal from an ordered slot is identical to element removal from an unordered slot since removing a specific element from an ordered slot does not alter the relative position of the other elements in the slot.

The precondition requires that a derived slot is not being modified and that the element must exist in the slot:

1. $\neg$derived(property(s))

2. $e \in$ elements(s)

The postcondition:

1. $(\forall t \in S \mid t \subseteq s \Rightarrow \text{elements}(t) = \text{elements}(t') \cup \{e\} \wedge e \notin \text{elements}(t'))$

2. $(\forall t \in S \mid s \subset t \wedge \neg(\exists m \in S \mid m \subset t \wedge m \,||\, s \wedge e \in \text{elements}(m)) \Rightarrow \text{elements}(t) = \text{elements}(t') \cup \{e\} \wedge e \notin \text{elements}(t'))$

Both clauses in the postcondition are interesting. The first clause states that a removal from a slot triggers a removal from any subset, so that the ISR can hold. This can be contrasted with the insertion operation, which does not modify any subsets. This contrast is perhaps a bit surprising; the lesson learned is that subsetting, at least in our semantics, is not as straightforward as one perhaps would hope and that exploiting subsetting successfully in modeling and metamodeling requires education and experience.

The second clause states that a removal from a slot triggers a (conditional) removal from any superset. An interesting feature of the clause is shown in Figure 17. If we have an initial setting as in case (1) and remove a from z, the clause requires that a is removed from x as shown in case (2), although this is not necessary to maintain model consistency. However, we believe that this feature is the intended usage by the developer. Inserting into a subset triggers insertion in all supersets, and so dually a removal from a subset ought to trigger a removal from all supersets. Thus it feels convenient to require this behavior. A similar chain of reasoning has been reported by Markus Scheidgen [21].

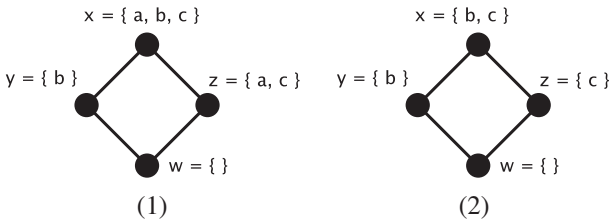


Figure 17: Removing a from an Unordered Slot z

As an example where the rather complicated formula in the second clause is necessary, consider Figure 18 with the initial setting as in case (1). Assume we wish to remove a from y. A simplistic removal of a from supersets and subsets would leave x without a, but z with a intact, violating the ISR, as shown in case (2). A second option would be to remove a also from z, as shown in case (3), but our opinion is that this "snowball effect" of removing a reduces the usefulness of subsets; slot y should affect slot z as little as possible, since they are not comparable in the Hasse diagram. Our postcondition ensures that a must be removed from w and y, but not from x, because z still contains a; this is seen in case (4).

5 An Implementation of Edit Operations for Models

In this Section, we give implementations for element creation and deletion as well as elements insertion into and removal from an unordered or ordered slot.

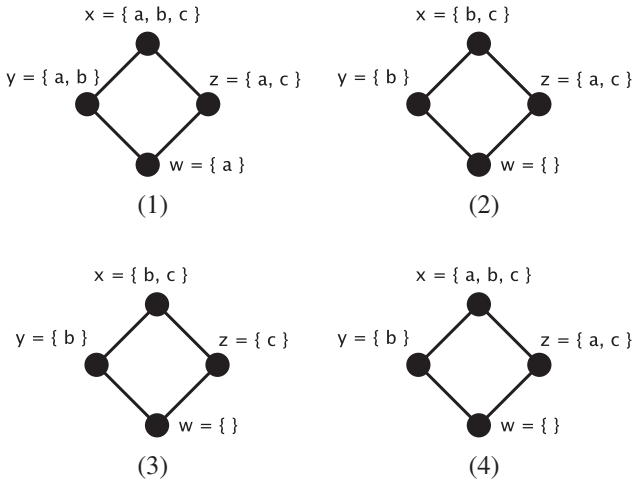


Figure 18: Different Scenarios for Removing a from an Unordered Slot y

5.1 Element Creation and Deletion

Element creation can be defined by inserting a new element into the set E and correctly updating the various functions that comprise the models. The operation is shown in Figure 19. We assume there is a programming language-dependent way to create a new element and new slots.

$$
\begin{array}{l}
\text{create}(M, c) := \\
\quad M = E, \text{type}, \text{slots}, S, \text{property}, \text{elements} \\
\quad \text{Create an element } e. \\
\quad \text{type}' := \text{type}[e \to c] \\
\quad E' := E \cup \{e\} \\
\quad \text{property}' := \text{property} \\
\qquad\qquad \cup \{ \text{ create a slot } s \text{ and return } s \to p \mid p \in P \wedge \text{owner}(p) \subseteq_c c \} \\
\quad S' := S \cup (\text{dom property}' \setminus \text{dom property}) \\
\quad \text{slots}' := \text{slots} \cup \{e \to s \mid s \in S' \setminus S\} \\
\quad \text{elements}' := \text{elements} \cup \{s \to [\,] \mid s \in S' \setminus S \wedge \text{ordered}(\text{property}'(s))\} \\
\qquad\qquad\qquad\qquad \cup \{s \to \{\,\} \mid s \in S' \setminus S \wedge \neg\text{ordered}(\text{property}'(s))\} \\
\quad \text{return } ((E', \text{type}', \text{slots}', S', \text{property}', \text{elements}'), e)
\end{array}
$$

Figure 19: Implementation of Creating a New Model Element

The implementation for element deletion is given in Figure 20. Here, $G \triangleleft F$ restricts the function F to the domain of a set G, i.e., $G \triangleleft F \stackrel{\text{def}}{=} \{x \to y \mid x \in G \wedge x \to y \in F\}$.

5.2 Insertion into an Unordered Slot

In the implementation given in Figure 21, an expression $F[G]$ denotes a copy of the function F with the domain and values modified as given by the set comprehension G:

$F[G] \stackrel{\text{def}}{=} \{x \to y \mid x \to y \in X \vee (x \to y \notin G \wedge x \to y \in F)\}$

$$
\begin{array}{l}
\text{delete}(M,d) := \\
\quad M = E,\text{type},\text{slots},S,\text{property},\text{elements} \\
\quad \text{type}' := \text{type} \setminus \{e \to c \mid e = d \wedge e \to c \in \text{type}\} \\
\quad E' := E \setminus \{d\} \\
\quad \text{property}' := \text{property} \setminus \{s \to p \mid \text{owner}(s) = d \wedge s \to p \in \text{property}\} \\
\quad S' := S \setminus (\text{dom property} \setminus \text{dom property}') \\
\quad \text{slots}' := \text{slots} \setminus \{e \to s \mid e = d \wedge e \to s \in S\} \\
\quad \text{elements}' := S' \lhd \text{elements} \\
\quad \text{return } (E',\text{type}',\text{slots}',S',\text{property}',\text{elements}')
\end{array}
$$

Figure 20: Implementation of Deleting a Model Element

$$
\begin{array}{l}
\text{insert}(M,s,e) := \\
\quad M = (E,\text{type},\text{slots},S,\text{property},\text{elements}) \\
\quad \text{elements}' := \text{elements}[\,\{x \to \text{elements}(x) \cup \{e\} \mid x \in S \wedge s \subseteq x\}\,] \\
\quad \text{return } (E,\text{type},\text{slots},S,\text{property},\text{elements}')
\end{array}
$$

Figure 21: Implementation of Element Insertion into an Unordered Slot

5.3 Insertion into an Ordered Slot

As we described in Section 4.4, inserting an element into an ordered slot is the most complicated operation on slots. The precondition could only tell us whether or not there is at least one solution, not what the exact combination of indices in different slots should be for a particular solution. Naturally, we must avoid the combinations that do not maintain the ISR.

Hence, we believe that the notion of an *insertion strategy* is important. Depending on the effect the developer wishes to obtain, such a strategy will mechanically calculate a particular solution and execute the actual insertion operation. At the moment we use only one strategy, that of always using the last index position possible.

The context in which the insertion strategy has to work is the final function F when T has been exhausted, as can be seen in Figure 15.

5.3.1 Last Index

Our implementation assumes that a correct combination of indices occurs if we always choose the last index (i.e. w of $F(t) = [v..w]$ for a slot t). This has worked perfectly in our experiments. Given our assumption, the implementation in Figure 22 is simple.

$$
\begin{array}{l}
\text{insert}(M,s,e,i) \\
\quad \text{Calculate the final } F \text{ as in Figure 15.} \\
\quad M = (E,\text{type},\text{slots},S,\text{property},\text{elements}) \\
\quad \text{elements}' := \text{elements}[\{t \mapsto \text{elements}(t)[0:w] \lhd [e] \lhd \text{elements}(t)[w:\#t] \\
\qquad\qquad\qquad\qquad \mid t \in S \wedge s \subseteq t \wedge e \notin \text{elements}(t) \wedge [v..w] = F(t)\}] \\
\quad \text{return } (E,\text{type},\text{slots},S,\text{property},\text{elements}')
\end{array}
$$

Figure 22: Implementation of the Insert Operation for Ordered Sets, Using the Last Index Strategy

For sequences, $\lhd$ denotes sequence concatenation and $t[a:b]$ denotes the sequence of elements $t[a],\ldots,t[b-1]$.

5.4 Removal from a Slot

The implementation for the removal of an element for an ordered or unordered slot is shown in Figure 23.

$$
\begin{array}{l}
\text{remove}(M, s, e) := \\
\quad M = (E, \text{type}, \text{slots}, S, \text{property}, \text{elements}) \\
\quad \text{elements}' := \\
\qquad \text{elements}[\,\{t \rightarrow \text{elements}(t) \setminus \{e\} \mid t \subseteq s\} \\
\qquad\qquad \cup \{t \rightarrow \text{elements}(t) \setminus \{e\} \mid s \subset t \\
\qquad\qquad\quad \wedge \neg(\exists m \mid m \subset t \wedge m \,||\, s \wedge e \in \text{elements}(m))\}\,] \\
\quad \text{return } (E, \text{type}, \text{slots}, S, \text{property}, \text{elements}')
\end{array}
$$

Figure 23: Implementation of Element Removal from a Slot

6 Conclusions, Related and Future Work

There are several new property characteristics described in MOF 2.0: subsets, (derived) unions and redefinitions. However, these standards do not describe these concepts in detail, not even informally, and therefore cannot be applied in practice. In this article, we have first described a simple formalization of metamodels and models and then presented basic operations for element creation and deletion and slot modification, taking into account subsets and derived unions. We have given usage examples where subsetting provides a new, fundamental approach to language extension. Several authors have used property subsetting informally, almost always referring to covariant specialization. Formalization of covariance leads to a different result than the one presented in this paper. Both subsetting and covariance specialization have their uses, however, and are thus complementing rather than competing constructs.

There are some limitations in the work presented in this article. Subsetting as proposed is restricted to slots with unique elements. Slots where the same element can occur several times (bags) are not considered.

Also, in this paper, we assume that a subset property should have the same ordering characteristic as its union property. However, we notice that it is also possibly to mix these characteristics such that an ordered slot may be a subset of an unordered slot, as we did in Fig. 7. The extension is trivial since it weakens the precondition because we do not need to maintain any indices in the unordered slot. The UML 2.0 Infrastructure uses this in the association between *association* and *memberEnd* in Figure 73 of [18].

However, the opposite case where an unordered property subsets an ordered property is problematic. Insertion into an ordered slot requires an index, but the initial insertion into the unordered slot does not tell which index or indices to use in any supersets which are ordered. We do not see any benefits in pursuing semantics for this construct. However, it must be noted that Figure 75 of [18] does show an example where an ordered property is subset by an unordered one. We believe this example, which is not part of the the UML 2.0 Infrastructure specification, to be erroneous.

Furthermore, we have not discussed metamodel evolution, where properties and classes are redefined, and how models must be updated accordingly, especially with respect to subsetting. We have followed a basic assumption that metamodels are static, but we should note that there are object-oriented frameworks where class updates or redefinitions are possible, for example Common Lisp [8].

6.1 Related Work

Several others have formalized the metamodel and model layers. For example, Thomas Baar has defined the CINV language [4] using a set-theoretic approach, but our approach is more general in that we also support e.g. generalizations. The benefits of a set-theoretic approach is that it avoids a metacircularity whereby one (partially) needs to understand the language to be able to learn the language. José Álvarez, Andy Evans and Paul Sammut describe such a static object-oriented metacircular modeling language [2], and the Metamodeling Language Calculus by Tony Clark, Andy Evans and Stuart Kent is another very sophisticated one. As such our basic framework does not describe anything novel.

Our contribution comes from the definitions of property subsets, which neither metamodeling nor traditional object/class language descriptions explain. Several authors use association inheritance without defining exact semantics, and some say that it denotes covariance. An example of this covariant specialization is the multilevel metamodeling technique called VPM by Varro and Pataricza [22], which also limits itself to single inheritance. We argue that property subsetting is not the same concept as covariant specialization, and requires different semantics.

Akehurst, Kent and Patrascoiu presented in [1] the structure of a metamodel and its semantics as an example of how to define model transformations using relations in OCL. However, they do not discuss the subset and union of properties.

Carsten Amelunxen, Tobias Rötschke and Andy Schürr have created the MOFLON tool [3] inside the Fujaba framework [14] which claims to support subsetting, but no description of the formal semantics they use is included. It is not clear if their tool works in the context of subsets between ordered slots, or with diamond inheritance with subsetting.

Markus Scheidgen presents an interesting discussion of the semantics of subsets in the context of creating an implementation of MOF 2.0 in [21]. To our knowledge, this has been so far the most thorough attempt to formalize subsetting. The approach is slightly different in that a slot modification creates an *update graph* of slots, so that a later modification at some other slot in the update graph actually updates all the associated slots. The actual operational semantics are unfortunately not described in detail. In comparison, we do not have to create or maintain any update graphs. Furthermore, our contribution not only discusses but also defines pre- and postconditions and implementations for the operations for ordered and unordered sets. It is also not clear if the work by Scheidgen supports diamond subsets or ordered sets, both of which are used in e.g. the UML 2.0 Infrastructure. However, our semantics are different and it is not clear which formalization is better suited for modeling purposes.

Unfortunately, we know of no tools that support subsets as extensively as proposed in this article. At the time of writing, the Eclipse EMF model repository does not implement subsets, although the feature is being planned.

6.2 Future Work

There are several different theoretical tasks for future work. The foremost task is to prove the correctness of the pre- and postconditions as well as the implementations. Second, throughout this paper, we have been clear that the elements in a slot are an unordered or ordered set. It might be of interest to formalize the framework for *bagness*, whereby a slot may contain the same element several times. It is fairly straightforward to extend this framework for unordered bags, even with subsetting, but our initial experiments with ordered bags and subsetting have not been as successful.

Third, we might wish to incorporate covariant specialization into the framework,

as so many authors are implicitly using covariance, and experience from the object-oriented community is that covariance is a very interesting and useful concept.

Fourth, the MOF characteristic of redefinition is still fairly arbitrary, badly defined concept, and could certainly take advantage of a more rigorous definition.

We believe that the formalization presented in this article can be implemented in a straightforward manner in a model repository. We plan to use our new definitions in implementing the UML 2.0 metamodel with diagram editors in our open source modeling tool called Coral. It can be downloaded from `http://mde.abo.fi`. In doing so, we strive to acquire experience in using subsets and derived unions in large models.

In conclusion, we consider that this work is important because there is an imminent need in the modeling community to standardize on one formalization of subsets and derived unions, so that tools implementing MOF 2.0 and UML 2.0 can be interoperable. The semantics described in this article is one proposal and we hope it spurs further interest and discussion. Furthermore, the idea of subsetting is intriguing, since it is a new construct for modeling relationships between classes and objects, and thereby brings a novel idea to the object-oriented community.

Acknowledgments

The authors would like to thank Patrick Sibelius for insightful discussions. Marcus Alanen would like to acknowledge the financial support of the Nokia Foundation.

References

[1] D. H. Akehurst, S. Kent, and O. Patrascoiu. A relational approach to defining and implementing transformations between metamodels. *Springer International Journal on Software and Systems Modeling*, 2(4), 2003.

[2] José Álvarez, Andy Evans, and Paul Sammut. MML and the Metamodel Architecture. In Jon Whittle, editor, *WTUML: Workshop on Transformation in UML 2001*, April 2001.

[3] Carsten Amelunxen, Tobias Rötschke, and Andy Schürr. Graph Transformations with MOF 2.0. In Holger Giese and Albert Zündorf, editors, *Fujaba Days 2005*, September 2005.

[4] Thomas Baar. Metamodels without Metacircularities. *L'Objet*, 9(4):95–114, 2003.

[5] J. Bézivin, E. Breton, G. Dupé, and P. Valduriez. The ATL Transformation-based Model Management Framework. Technical Report 03.08, IRIN Univesité de Nantes, 2003.

[6] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick, and Timothy J. Grose. *Eclipse Modeling Framework*. Addison Wesley Professional, August 2003.

[7] Giuseppe Castagna. Covariance and Contravariance: Conflict without a Cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[8] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp object system: an overview. In *European conference on object-oriented programming on ECOOP '87*, pages 151–170, London, UK, 1987. Springer-Verlag.

[9] EMF Development team. The Eclipse Modeling Framework. http://www.eclipse.org/emf.

[10] R. France and B. Rumpe. Domain specific modeling, Editorial. *Springer International Journal on Software and Systems Modeling*, 4(1), 2005.

[11] Anneke Kleppe, April 2003. Discussion on the mailing-list pumllist@cs.york.ac.uk.

[12] Barbara Liskov. Keynote address - data abstraction and hierarchy. *SIGPLAN Not.*, 23(5):17–34, 1988.

[13] Netbeans. Netbeans Metadata Repository (NMR). Available at `http://mdr.netbeans.org/`.

[14] Ulrich A. Nickel, Jörg Niere, and Albert Zündorf. Tool demonstration: The FUJABA environment. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, pages 742–745. ACM Press, 2000.

[15] OMG. Meta Object Facility, version 1.4, April 2002. Document formal/2002-04-03, available at `http://www.omg.org/`.

[16] OMG. MOF 2.0 Query / Views / Transformations RFP. OMG Document ad/02-04-10. Available at www.omg.org, 2002.

[17] OMG. Meta Object Facility (MOF) 2.0 Core Specification, October 2003. Document ptc/03-10-04, available at `http://www.omg.org/`.

[18] OMG. UML 2.0 Infrastructure Specification, September 2003. Document ptc/03-09-15, available at `http://www.omg.org/`.

[19] OMG. UML 2.0 Superstructure Specification, August 2003. Document ptc/03-08-02, available at `http://www.omg.org/`.

[20] OMG Architecture Board. Model Driven Architecture - A Technical Perspective. OMG Document ormsc/01-07-01. Available at www.omg.org, 2001.

[21] Markus Scheidgen. On Implementing MOF 2.0—New Features for Modelling Language Abstractions. July 2005. Available at `http://www.informatik.hu-berlin.de/~scheidge/`.

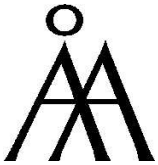
[22] Dániel Varró and András Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling*, 2(3):187–210, October 2003.

[23] Andreas Winter, Bernt Kullbach, and Volker Riediger. An Overview of the GXL Graph Exchange Language. In *Revised Lectures on Software Visualization, International Seminar*, pages 324–336, London, UK, 2002. Springer-Verlag.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1653-0
ISSN 1239-1891